REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* 30-09-2012	**2. REPORT TYPE** Final Performance Report	**3. DATES COVERED** *(From - To)* March 1, 2009 - June 30, 2012

4. TITLE AND SUBTITLE Information Foraging Theory in Software Maintenance	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER FA9550-09-1-0213
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Burnett, Margaret M	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Oregon State University, Corvallis, OR 97331	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Office of Scientific Research, 875 N Randolph St, Arlington, VA 22203, Dr. Robert Bonneau/RSL	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) AFRL-OSR-VA-TR-2012-1210

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approve for Public Release: Distribution Unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This grant provides a theoretical foundation as to how to support programmers' navigation. We develop a theory of information foraging for software maintenance. Then, to test the theory's validity, generality, and scope, we build predictive models, and use them for empirical lab investigations to evaluate our progress. Finally, we develop library modules and tools for use to empirically investigate real-world settings. The resulting theoretical foundation can replace practices of building software maintenance tools ad hoc, enabling principled progress in supporting programmers who maintain today's complex software.

15. SUBJECT TERMS
information foraging theory, software debugging, programmer navigation

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Margaret M. Burnett
U	U	U	U	19	**19b. TELEPHONE NUMBER** *(Include area code)* 541-737-2539

Reset

Final Performance Report

Air Force Office of Scientific Research

Systems and Software Program
Dr. Robert Bonneau, Program Manager

Grant:
FA9550-09-1-0213

Reporting Period:
March 1, 2009 - June 30, 2012

Project Title:
Information Foraging Theory in Software Maintenance

Principle Investigator:

Margaret Burnett
Professor of Computer Science
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR 97331
Phone: 541-737-2539
Email: burnett@eecs.oregonstate.edu

1. Overview: Objectives and Accomplishments

The purpose of this grant was to develop a theory of information foraging for software maintenance. The original grant had six objectives that are summarized in Table 1. The original objectives were 1, 2, 3, 4, 5a, and 6a. As the table shows, the PI and then-Program Manager Dr. Luginbuhl eventually decided to replace Objective 4 by Objectives 5b and 6b, in order to leverage the potential synergy with the Large-Scale Cognitive Modeling effort supported by Grant FA9550-10-1-0326.

We have accomplished all of these objectives, and our work is already making an impact. Since this grant began, we have produced 12 publications, and these publications have already accumulated 87 citations.

Table 1 summarizes the accomplishments by objective, and the next several sections detail the accomplishments for each objective.

Table 1: Themes, Objectives, and Publications of this Grant. (Deleted objective is struck out, new objectives are italicized.)

Theme	Objectives accomplished	Publications
Define the theory	Objective 1: Define an information foraging theory (IFT) for software maintenance	Lawrance et al. (to appear) Lawrance et al. 2010 Ko et al. 2012 Burnett et al. 2011 Piorkowski et al. 2012b (in review)
Develop and empirically evaluate computational models	Objective 2: Develop predictive computational models for programmer navigation	Lawrance et al. 2010 Lawrance et al. (to appear) Piorkowski et al. 2011
	Objective 3: Evaluate predictive computational models for programmer navigation	
(Replaced by Objectives 5b and 6b)	~~Objective 4: Evaluate IFT on perfective software maintenance~~	N/A
Develop and empirically evaluate software modules for tools	Objective 5a: Develop software modules to apply or support IFT in navigation tools	Piorkowski et al. 2012a Piorkowski et al. 2011 Piorkowski et al. 2010 Fleming et al. (to appear) Piorkowski et al. 2012b (in review)
	Objective 6a: Evaluate software tools that apply or support IFT	
Apply IFT to software tasks in Cognitive Modeling	*Objective 5b: Same as Objective 5a, but for cognitive modeling tools*	Bogart et al. 2010 Bogart et al. 2012
	Objective 6b: Same as Objective 6a, but for cognitive modeling tools	

2. Detailed Activities and Findings

2.1 Objective 1: Define an information foraging theory (IFT) for software maintenance

Past theories of how programmers debug were developed in the context of very simple programming environments and very small programs. Further, they rely on complex mental constructs that offer little practical advice to builders of software engineering tools. In this research, we reconsidered how programmers go about debugging in large collections of source code using a modern programming environment. Building on the pioneering work of Pirolli and his colleagues [Chi et al. 2001, Fu et al. 2007, Pirolli 1997, Pirolli and Card 1998, Pirolli and Card 1999], we developed an information foraging theory for the context of debugging that treats programmer navigation as being analogous to a predator following scent to find prey in the wild. The theory proposes that constructs like "scent" emanating from cues in the environment and "topology" of the information space provide enough information to describe and predict programmer navigation during debugging, without reference to in-the-head states of programmer cognition.

We began by defining our IFT variant, focusing particularly on its constructs. We then performed empirical analyses of programmers' behaviors according to these IFT constructs to validate the constructs and utility of our theory.

2.1.1 The theory's constructs, and how to operationalize them

To handle the realm of debugging, we built upon the original information foraging constructs, resulting in the following definitions [Lawrance et al. 2009, 2010]:

- *Predator*: The programmer who is debugging.
- *Prey*: What the programmer seeks to know to reveal the changes that must be made to fix the bug. Furthermore, any information that the programmer seeks to achieve the goal also constitutes a form of prey.
- Information *patches*: Localities in the source code, related documents, and displays that may contain prey.
- *Proximal cues*: Words, objects, and perceptible runtime behaviors in the programming environment that suggest scent relative to the distal prey. Cues act as signposts to prey. For example, words in the source code, including comments, constitute a type of cue.
- *Information scent*: The perceived likelihood of a cue leading to prey, either directly or indirectly. Scent is a measure, and scent from one cue can be compared to the scent of other cues. Unlike cues, scent exists only in the programmer's head. This definition is consistent with Chi, Pirolli, et al.'s definition of information scent as "the subjective sense of value and cost of accessing [information] based on perceptual cues" [Chi et al. 2001].
- *Topology*: The collection of paths through the source code, related documents, and displays through which the programmer can navigate.

To allow computational models of the theory, we developed the following operational definitions. Some constructs have self-evident operationalizations: we simply operationalize the prey construct as places in the code where changes must be made to fix the bug, the predator construct as a programmer, and the patch construct as localities in the source code, such as Java methods, classes, and packages.

The notions of topology, cues, and scent are not as simple. We operationalize these constructs as follows:

- *Topology*: A directed graph with vertices representing elements of the source code (e.g., classes, methods) and of the environment (e.g., class labels enumerated in a class hierarchy browser), and with edges representing navigable links between the elements.

- *Link*: A connection between two nodes in the topology that allows the programmer to traverse the connection at a cost of just one click. For example, in the Eclipse editor, a method invocation can be clicked on to open the associated method definition. Hence, Eclipse provides links from method calls to definitions that the predator can navigate with one click. In this case, a method call is the *source* of a link and the associated definition is the *destination*. (Note that links are environment and context dependent.)
- *Proximal cues*: Words located near the source of a link. For example, consider the following line of code opened in the Eclipse editor: "`System.out.println(someData);`". The identifier `println` is the source of a method-invocation link, and the words `system`, `out`, `println`, and `someData` are proximal cues that engender scent about potential prey at the other end of the link.
- *Scent*: Word similarity between the bug report (description of the prey) and proximal cues. That is, a set of cues comprising words that appear frequently in the bug report will engender strong scent in the mind of the predator.

The measure of scent warrants further explanation. Information scent is the programmer's (imperfect) perception of the value (relatedness) of information (as in Pirolli's information foraging research on web searching [Pirolli 1997]). To computationally approximate information scent, we compute word similarity between the description of the prey (e.g., bug-report text) and the proximal cues in the source code by applying cosine similarity to a vector space IR model. Note that this operational definition is the model's *approximation* of scent; the true measure of scent exists only in the programmer's head.

Computing this approximation of scent is a three-step process. First, we preprocess the source-code text and bug report. We tokenize words so that camel case identifiers (e.g., "`NewsItem.getSafeXMLFeedURL()`") are split into their constituent words (e.g., "news," "item," "get," "safe," "xml," "feed," and "url"). Furthermore, we apply the Porter stemming algorithm on the constituent words[1] and filter out Java reserved words (e.g., `public`, `static`, and `void`) and English stopwords (e.g., "the," "to," "be," "or," and "not").

Second, we weigh terms in files of source code according to the commonly used *tf-idf* formula [Baeza-Yates et al. 1999], which we compute as follows:

$$w_{i,j} = f_{i,j} \times idf_i$$

where $f_{i,j} = \frac{freq_{i,j}}{\max_{\forall v} freq_{v,j}}$ and $idf_i = \log \frac{N}{n_i}$

Here, $f_{i,j}$ is the frequency of word i in document j (normalized with respect to the most frequently occurring word v in a document), idf_i is the inverse document frequency, and $w_{i,j}$ is the weight of word i in document d_j.

Third, we compute the inter-word correlation between proximal cues in source files and the text of a bug report using cosine similarity, a measure commonly used in

[1] Porter stemming is simple and efficient, but, like all stemming algorithms, it can stem words erroneously (i.e., producing different stems for different forms of the same words, or producing the same stem for different words) [Baeza-Yates et al. 1999].

information retrieval systems [Baeza-Yates et al. 1999]. We compute cosine similarity as follows:

$$sim(d_j, q) = \frac{d_j \cdot q}{\|d_j\| \times \|q\|} = \frac{\sum_{i=1}^{t} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^{2}} \times \sqrt{\sum_{i=1}^{t} w_{i,q}^{2}}}$$

where $w_{i,q}$ is the weight of word i in the bug report text (i.e., the query in the terminology of [Baeza-Yates et al. 1999]).

2.1.2 Evaluating the theory's constructs

We then performed a qualitative analysis [Lawrence et al. 2009; Lawrance et al. to appear] of lab study data that we had collected during preliminary work. The data consisted of videos and logs of 12 IBM programmers' navigations while debugging an open-source project we asked them to work on. The purpose of the analysis was to attempt to apply the constructs rigorously to the behavior of real programmers, and also to compare use of these constructs to use of the more traditional construct of hypotheses. Our primary research questions for this analysis were:

- RQ1: The literature emphasizes the importance of programmer hypotheses in understanding debugging behavior. Is there reason to expect a scent-based model of debugging navigation will succeed if it does not explicitly handle programmers' hypothesis processing?
- RQ2: Debugging involves a variety of activities, such as locating the fault, fixing the fault, and verifying the fix. When do developers engage in scent and hypothesis processing with respect to these activities?
- RQ3: Programmers navigate through a variety of artifacts, such as source code, documentation, and email. Where do programmers navigate during debugging with respect to these artifacts, and in which artifacts do programmers exhibit scent seeking and hypothesis processing?

Our results for each of these research questions were:

- RQ1: The way participants worked with scent was consistent with information foraging theory. The participants verbalized activities related to scent about four times as often as (non-scent) hypotheses. The relationships between scent and some types of hypotheses may have contributed to the effectiveness of scent as a predictor.
- RQ2: In the six debugging modes in our participants' data, scent following was pervasive in all six of them, whereas (non-scent) hypotheses were mostly concentrated in just one of them, the predominant "fix" phase. This finding also helps to explain why scent was so effective at predicting programmer navigation.
- RQ3: The biggest "trigger" for scent following was the source code itself, but other triggers included the bug report, runtime behavior, and additional resources such as web pages and input files. These findings suggest, first, that operationalization of the scent construct using static analysis of source code alone can produce reasonably accurate predictions and, second, that even greater accuracy may be possible if a model includes these additional data sources.

2.1.3 Evaluating the theory's utility

One desirable attribute of a theory is its utility, i.e., its ability to generalize or gain understanding in ways not easily done before. To evaluate the utility of our theory, we turned our attend to one of the least studied areas of Information Foraging Theory, namely *diet*: the collection of information foragers choose to seek. In the domain of software engineering, there has been a little empirical work in that area, but because it was atheoretic, generalizing it has not been easily done. Examples of the kinds of questions we investigated were: do foragers choose solely based on cost, or do they stubbornly pursue certain diets regardless of cost? Do their debugging strategies vary with their diets? [Piorkowski et al. 2012b].

To investigate "what" and "how" questions like these for the domain of software debugging, we analyzed 9 professional developers' foraging goals, goal patterns, and strategies. To bring our work together with the software engineering community's, we began with a code set we based on the Information Goal code set on Sillito et al.'s empirically based taxonomy of 44 questions programmers ask, which Sillito et al. had grouped into four types [Sillito et al. 2006]. We also coded the strategies these professional programmers used, according to the categorization of Grigoreanu et al. [2010]. Finally, we analyzed goal sequences for information foraging "patterns", and analyzed strategies by information foraging activity type. These analyses together produced insights into which information foraging activities and strategies programmers were using to target their different information goal types, and how the foraging patterns related to these results. Table 2 shows the results for the top strategies.

Table 2: Top strategies, their IFT activity category, which goal types professional programmers used them for, and in what kinds of foraging patterns [Piorkowski et al. 2012b].

Strategy	How many used it?	Top strategy for...			
		... which participants	... which IFT category	... which Goal Type	... which Patterns
Within-Patch Strategies					
Spatial	all 9	P9	Within	2-Build	Pyramid
Code Inspect.	all 9	-	-	-	-
Between-Patch Strategies					
Control Flow	all 9	-	Between	-	Restart
Feedback Follow.	all 9	-	-	-	-
Enrichment Strategies					
Code Search	5	P3, P6, P7	-	1-initial	Repeat, Oscillate, Stairstep
Testing	8	P2, P5, P8, P10, P11	Enrich.	3-group, 4-groups	-

One surprising finding was the sheer amount of foraging: Participants spent 50% of their debugging time foraging for information to satisfy their diets. This suggests that better support for programmers' foraging efforts during debugging could potentially bring huge reductions in debugging costs.

Participants' goals often progressed in specific patterns: Just five such patterns covered 58% of the participants' foraging. However, the patterns they mostly followed were not those predicted by previous literature. Specifically, use of the orderly Stairstep, Restart, and Pyramid patterns were relatively uncommon with less than 22% of the data following such patterns. The majority of their foraging instead fell into long stretches of foraging for goals of a single type (the Repeat pattern).

The participants' sometimes stubborn pursuit of particular information goals—tolerating very high costs even when their efforts showed only meager promise of delivering the needed dietary goal—highlights an important difference in the software domain versus other foraging domains: Programmers' dietary needs are often very specific. For an information forager on the Web, one dictionary page is often as good as another. But for a programmer trying to fix a bug, only very particular information about very specific code locations will help them in their task. This high dietary selectiveness in this domain may explain the high costs programmers were sometimes willing to pay.

2.2 Objectives 2 and 3: Develop and evaluate predictive computational models for programmer navigation

For Objectives 2 and 3, we incorporated the operationalizations from Section 2.1 into computational models, to evaluate the validity of IFT's predictiveness of programmers' navigations. We verified these computational models' predictions against logs of programmers performing debugging. We built three such models: PFIS [Lawrance et al. 2009], PFIS2 [Lawrance et al. 2010a] and PFIS3 [Piorkowski et al. 2011].

2.2.1 PFIS: Programmer Flow for Information Scent

Our first model was called PFIS (Programmer Flow for Information Scent) that predicts programmer method navigation [Lawrence et al. 2009; Lawrence et al. 2010b]. PFIS calculates the probability that a programmer will follow a particular "link" from one class or method in the source code to another, given a specific information need.

More specifically, the model constructs a *source topology graph* T of the parts of the system that the developer has seen so far. The vertices of T represent patches (e.g., methods, packages, classes, and variables), and the edges represent between-patch links (e.g., "has-a" and "calls-a" relationships between these elements, and within-file adjacency relationships from the code). Thus, the set of classes A includes every class referenced in a file that developer has opened so far. Similarly, the set of methods M, set of variables I, and set of packages P include every method, variable, and package, respectively. For every element $e \in A \cup M \cup I \cup P$, the graph T contains a unique vertex that maps to e. T contains an edge between two elements e_a and e_b if e_a calls e_b, e_a has e_b or e_a and e_b are both methods and are adjacent in a source file.

The graph T also includes words (the *information features* in the model) from the source files. Formally, D is the set of words that occur in all the source files that the developer has opened so far. A unique vertex is added to T for each word $d \in D$. Edges are added such that a word node w is connected to a non-word node x if the text associated with x (be it name, definition, or Javadoc) contains d.

The model uses the above data structure to combine two factors: source-code topology and method-text similarity. It propagates the weights of each, using an algorithm based on *spreading activation*. Formally, each vertex in T has an activation value, initially 0. The activation of the current method vertex m is set to 1. The model "pumps" activation three times, spreading the values to adjacent nodes. The resulting nodes are then ranked according to weight, returning an ordered list of predictions as to which method the developer will visit next.

Our evaluation of PFIS showed that the model was able to predict the classes that programmers navigated to better than a model that predicted visitations based on text similarity (calculated by cosine similarity of text in methods) alone for some tasks (see the ROC curve in Figure 1). The good performance of this first model led to further developments of PFIS into PFIS2 and PFIS3.

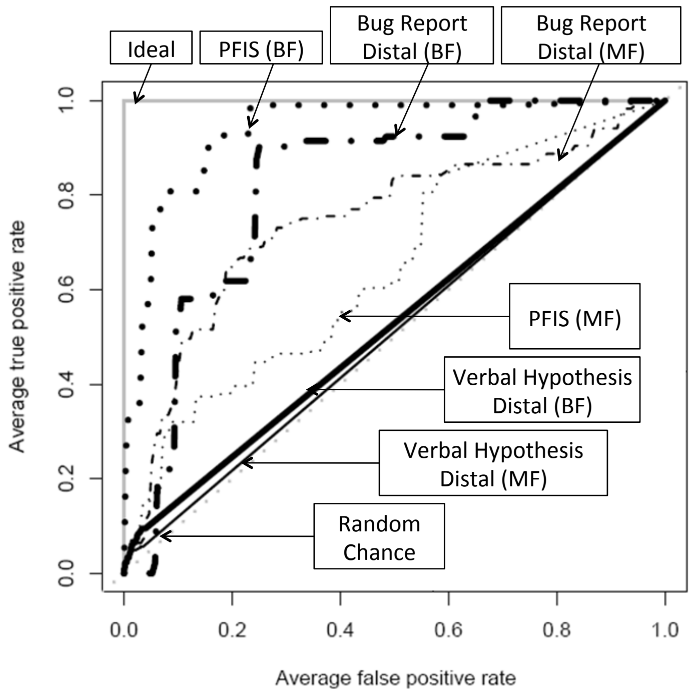

Figure 1: Verbal hypothesis ROC curve: Ability to predict classes to which participants navigated directly after they expressed a hypothesis.

2.2.2 PFIS2: Adding reactivity

Computational models of other variants of information foraging theory, as well as our own PFIS model, assumed a relatively "fixed" goal; i.e., that the description of prey up front would be sufficient for the model's understanding of the predators' goal. PFIS also had a simplifying assumption that the source code never changes, which is of course not realistic for debugging situations.

Relaxing both of these constraints, we developed a new model called PFIS2 that models information seeking when developer's goals evolve over time [Lawrance et al. 2010a]. We then evaluated variants of this model in a field study that analyzed programmers' daily navigations over a seven-month period. Our results were that PFIS2 predicted users' navigation remarkably well, even though the goals of navigation, and even the information landscape itself was changing markedly during the pursuit of information.

We evaluated PFIS2's suitability for modeling real-world foraging by comparing how well PFIS2 could predict where these programmers really navigated. Our empirical results revealed that:

- PFIS2 accurately predicted our participants' navigation. The most successful PFIS2 variant (PFIS2-AllScent) achieved a median prediction rank of 3, and even predicted our participants' navigations as its first choice 27% of the time. This occurred even in the absence of explicit descriptions of the prey such as bug reports. (See Figure 2 and Figure 3.)

- PFIS2's success was tied to course correction. Its incremental notions of prey allowed it to recover from big surprises very quickly (median: only one navigation for the two best variants).

2.2.3 PFIS3: A parameterized model

In light of these results, we began a number of comparisons to isolate specific factors that could be contributing to PFIS2's past (and future) success. We went about this investigation by parameterizing our model. We call the new parameterized model PFIS3. Model parameters include *recency*, which assigns higher values to methods that the programmer recently visited; *working set*, which assigns higher values to a group of methods that were visited most recently; *frequency*, which assigns higher values to methods that the programmer visits in total for the entire session; and *bug report similarity*, which assigns higher values to methods that are textually-similar to a bug report.

In addition, we tested combinations of these single-factors in a theoretical ideal model. Our optimal composite model combined two or more single-factor models and

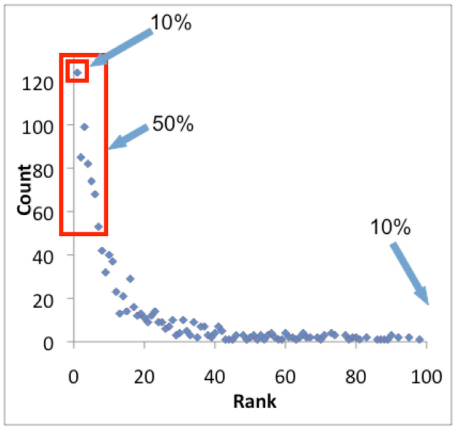

Figure 2: PFIS2 performance with bug reports

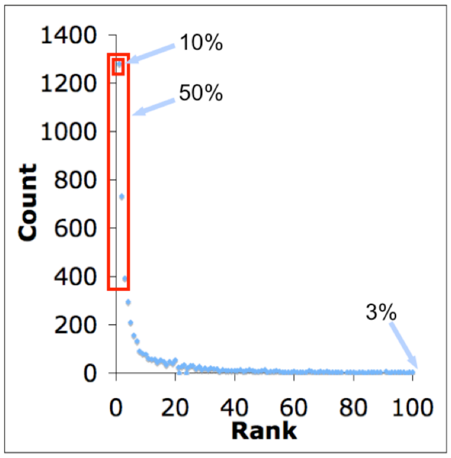

Figure 3: PFIS2 performance without bug reports. (Not shown: 27% of the time, programmers' navigations were the model's first choice.)

was considered to accurately predict a single navigation any time one of its single-factor models got a hit in the top 10 ranked results. Clearly, this model is not feasible, but was developed to see what a best-case scenario would look like.

We further tested each models' predictive accuracy using two different operationalizations of programmer navigation. The *click-based* operationalization, where a navigation was counted each time the text cursor of the editor was in a method, represented a navigation path that could be captured by a logging tool (Figure 4). The *view-based* operationalization coded methods all methods visible on the screen as navigations, thereby including many more methods that the click-based approach alone. These navigations had to be coded manually, but represented a more complete set of what the programmer saw when debugging (Figure 5).

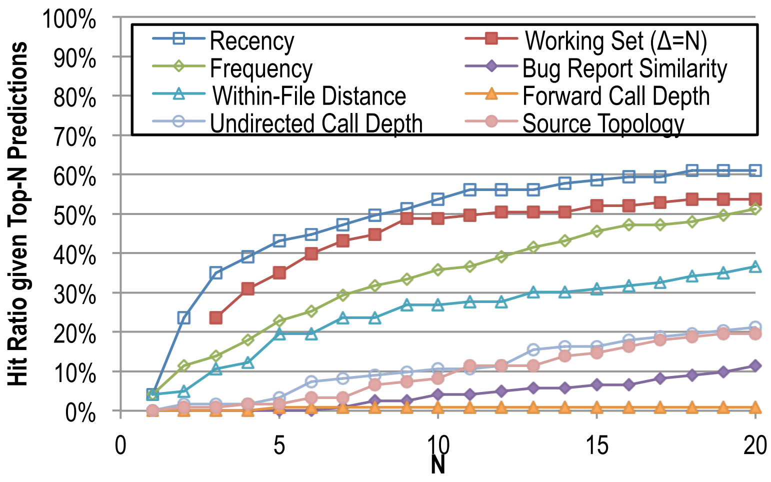

Figure 4: Accuracy for PFIS3 variants for click-based navigations

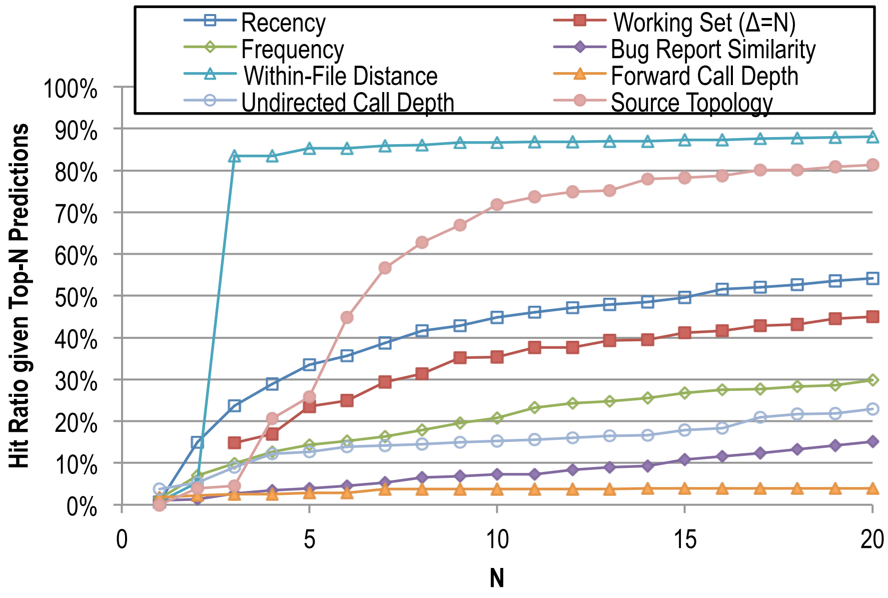

Figure 5: Accuracy for PFIS3 variants for view-based navigations

These figures reflect empirical comparisons against programmers' actual navigations. Overall in this investigation, we found that:

- Recency was the most accurate model for predicting click-based navigations.
- Scrolling actions were very common, and accounting for them with a view-based operationalization of navigation revealed much higher accuracy for Within-File Distance and lower relative accuracy for Recency than previously reported.
- Bug Report Similarity exhibited low accuracy all around—a surprising finding given that many approaches use bug reports as input.
- Our evaluation of optimally composed multi-factor models revealed the high

potential of composing Recency and Within-File Distance to enhance accuracy,

- Our spreading activation based multi-factor model, PFIS3+Recency, stood out as demonstrating consistent performance across both click- and view-based navigations.

2.3 Objective 5a and 6a: Develop and evaluate software modules to apply or support IFT

We have created three types of software tools to encourage the adoption of scent into navigation tools: a data-gathering tool in Eclipse, a method recommendation system for programmers in Eclipse, and a library of design patterns for tool builders.

2.3.1 PFIG: A tool to collect information foraging data

PFIG (programmer flow information gatherer) is a data collector within Eclipse that provides data for these models [Lawrance 2009]. The system can capture navigation events and upload them to a central database. Using this tool, future tool builders can gather programmer navigation data in the presence of their own Eclipse add-ons, to facilitate evaluations of their own software tools for navigation support.

2.3.2 PFIS-R: A navigation recommender based on PFIS3

We also built a prototype recommender system based on an extension of PFIS3. We call this PFIS3-based recommender tool PFIS-R. This recommender tool provides in real time a list of methods that it recommends a programmer consider navigating to, plus shortcuts and bookmarking features to facilitate efficient navigation to them.

Based on our empirical results of the predictiveness of various versions of PFIS3 (Section 2.2), the PFIS-R recommender tool incorporates three factors:

- *Text similarity.* Text within a method that was similar to the text of the method a developer is currently viewing is recommended highly compared to methods that are dissimilar. The text of a method are created as a stemmed “bag of words” with English and Java stopwords removed
- *Recency.* A method that was among the *n* most recently visited is weighted more heavily than a method that was not recently visited. The values used in this experiment were 1 and 10.
- *Code structure.* A method that is calls or is called by the method a developer is currently viewing is recommended highly compared to methods that do not refer to or are not referred by the current method.

To evaluate PFIS-R, we installed it in Eclipse, and then empirically investigated how 9 professional programmers from a large software company used different variants of PFIS-R when fixing an actual bug from an open-source project with almost 6,500 methods.

Our findings were:

- When considering the next 10 navigations, developers more frequently navigated to methods suggested by the recommender (whether using the tool or not) when PFIS-R was seeded with a history of only the previous navigation than when it was seeded with a history of the past 10. This was true even when we considered the next 1 navigation only.

- As developers progressed within the task, developers more frequently navigated to methods suggested by PFIS-R. This increase in performance suggests that the tool accurately models developers who were comfortable with the code base and the task.
- Programmers responded positively to the PFIS-R version of the recommender tool. They used it not only to discover new methods, but also to efficiently navigate to methods that they had visited before.

This work highlights the importance of incorporating recency into recommendations, and suggests that developers gain stronger scent as they progress throughout the task. The study also highlighted the relevance of semantic similarity as a method of recommending navigation. One implication of these findings is that information foraging theory may need to become more *adaptive*, changing in response to a developer learning about the code base that they are foraging in.

2.3.3 IFT design patterns

To enable software engineering researchers to design effective tools that use IFT principles, we surveyed numerous software maintenance tools in the software engineering literature, with the goal of showing how the theory can both demonstrate the commonality among those tools (to avoid continually reinventing certain aspects of them) and of producing practical design aids for tool builders. This work resulted in twelve *information foraging theory design patterns* [Fleming et al. 2012]. These patterns are summarized in Table 3.

Table 3: Summary of Information Foraging Theory design patterns for tool builders [Fleming et al. 2012]

Patterns for identifying valuable prey and patches
Expertise recommender: Identify people who can help the developer with an information goal
Lexical similarity: Identify patches that contain valuable prey based on their lexical similarity to words that emit strong scent to the developer
Past aggregate behavior: Identify patches that contain valuable prey based on the past behavior of developers working on the project.
Personal working set: Identify patches that the developer will likely revisit based on how recently the developer accessed or modified patches.
Structural relatedness: Given a current patch in which the developer has indicated interest (e.g., by navigating to the patch and/or modifying the patch), this pattern seeks to identify other patches that contain valuable prey by looking at other patches that are structurally related to the current patch.
Task heuristic: Identify patches that contain valuable prey by leveraging knowledge about the specific task that the developer is performing, and in particular, about what information developers typically seek during the task.
Patterns for presenting prey
Community portal: To provide an information patch where multiple contributors can enrich the patch with information features, especially cues.
Cue decoration: To automatically change the appearance of a cue to attract more of the developer's attention or augment a cue with additional information to improve the accuracy of scent discerned by the developer.
Dashboard: To generate an information patch in which a developer can become aware of links that lead to continually changing information patches relevant to his or her work.
Filtering: To enable the developer to filter out irrelevant information features from a patch, reducing the cost of processing the patch.
Gather together: To enable a developer to assemble information features from disparate patches into a single patch, thus reducing the cost of navigating between those features.
Signpost: To support enrichment by enabling the developer to leave cues in the environment which, when seen later, will attract the developer's attention and/or engender accurate scent in the predator's mind.

2.4 Objectives 5b and 6b: Apply IFT to software tasks in cognitive modeling

To leverage synergy with another AFOSR project, with the encouragement of the Systems and Software Program Manager at the time (Dr. Luginbuhl), we adapted information foraging theory to a non-traditional software development domain: the exploration of cognitive model traces. This helps to establish the generality of the theory.

2.4.1 IFT in cognitive modeling

Cognitive modelers are psychologists, linguists, etc., who write computational models that simulate the human mind, in order to test theories about cognition. In our initial formative work with AFRL cognitive modelers [Bogart et al. 2010] we described several aspects of this endeavor that make it a particularly challenging foraging task, and also somewhat different from the way professional programmers navigate in more traditional types of Java-based software:

- Modelers are comparatively more interested in foraging through behavior traces rather than model "source code", since models tend to be small and parsimonious, but model behavior is less predictable. Unlike Java statements that typically have deterministic effects, the commands issued by cognitive model rules are mediated by complicated equations with noise parameters. Thus, modelers spend more time navigating behavior traces than navigating source code.

- Not all scents in Java IFT apply well to behavior traces. In particular "lexical similarity" makes less sense in this domain since the lists of numbers and string values found in a trace are often highly repetitive or uninformative in isolation. Instead, modelers' information seeking goals are *patterns*: sequences or juxtapositions of data, rather than individual words or numbers. Their scents must therefore relate to critical differences between those sought-out patterns.

2.4.2 Patches in cognitive modeling

Thus we needed to consider how to operationalize the IFT constructs differently to apply to this domain. Whereas *patches* in Java navigation tend to be methods or classes, in the cognitive modeling domain we defined them as distinct *visualizations* (graphical or textual/tabular/list) of trace data. In our work [Bogart et al. 2010], we defined "evaluation abstractions" as the abstract information need that the visualizations met (or did not meet, in many cases). In that work we found three broad categories of evaluation abstraction (see Figure 6):

- "Data" abstractions related information at a particular point in time: for example a diagram of a tree structure in memory at the end of a model run
- "Time" abstractions related information across time: for example a time series plot of model reaction times to many (simulated) stimuli presented to it in a run
- "Statistical" abstractions summed, averaged, or otherwise summarized many data items

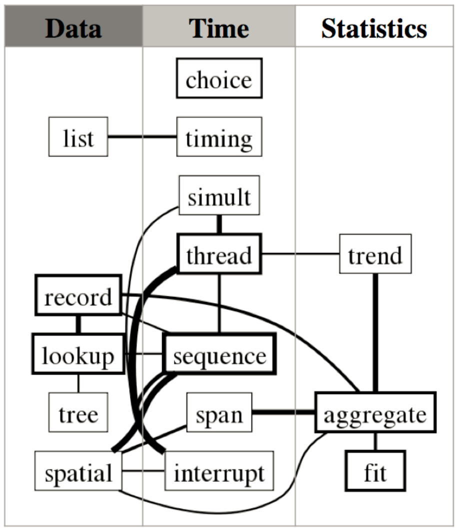

Figure 6: Three categories of abstract "patches" in analyzing behaviors of cognitive models. We observed several kinds of abstraction in each category. This figure denotes high co-occurrence with thicker lines.

2.4.3 Navigation in cognitive modeling

In an empirical study of how people navigated between their various views and visualizations while performing cognitive model debugging tasks, we were able to use this information foraging theory perspective to identify serious shortcomings in the tool set:

- There simply were no patches for many types of common abstractions that modelers verbalized an interest in. Modelers had to navigate to a series of patches and remember or write down information to assemble the evaluation abstraction that interested them.
- Navigation affordances between patches bore little relationship to the paths modelers needed and wanted to follow. Some of ACT-R's native trace visualizations are useful for very particular kinds of questions, but there is no way to navigate between them when modelers sought answers that crossed boundaries between them. So the "navigation" might involve restarting the model with identical parameters and somehow finding one's place again in the new view.
- These expensive navigations in the middle of answering a single "evaluation abstraction" cause modelers to forget the first half of the answer they'd already achieved.

To find out what navigations should be supported by cognitive model behavior analysis tools, we conducted a Wizard-of-Oz study [Bogart et al. 2012]. In this study we let modelers ask for any kind of visualization they could imagine, one after another, in pursuit of an answer to a question posed to them about a misbehaving model. After each request, the experimenter tried to give them a close approximation of that using a query tool customized for the experiment.

This Wizard-of-Oz experiment helped us confirm and expand on prior results about what patches should ideally be provided (Figure 7(a)). It also helped us find out, from the sequence modelers followed in pursuing questions among these patches, what the ideal navigation topology should be between hypothetical evaluation abstraction patches.

We characterized the results of this study by designing a descriptive model of these navigations in the form of an abstract syntax. The syntax encoded each possible query in the space of queries modelers made (Figure 7(b) and (c)), and we arranged the syntax such that high-importance navigations were represented by proportionally smaller changes to the syntax. The syntax can then serve as a representation of both patch and topology; the majority of user navigations within a hypothetical interface designed around the syntax are low-cost (Figure 7 (d)).

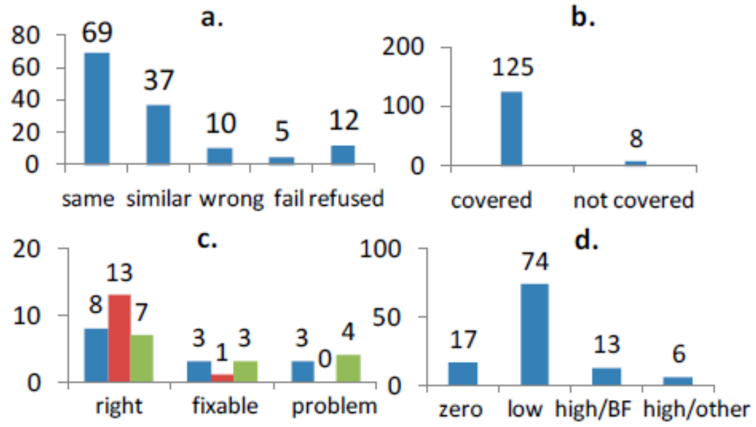

Figure 7: Validation results for the abstract syntax describing modeler navigation among evaluation abstraction patches. (a) How well the Wizard-of-Oz experimenter did at meeting participant requests in the "live" task during the experiment (b) How many of the abstractions in the study our abstract syntax covered (c) How well abstract syntax covered modelers needs in retrospect, judged by a panel of experts. (d) How expensive the navigations in the study would have been in a hypothetical interface designed around the abstract syntax.

We have turned the abstract syntax described above into an Eclipse plugin, adapted it for five different cognitive modeling languages. Going forward with the sister grant, which ends six months after the one we are reporting on here, we are currently evaluating it by running a field study of a model behavior analysis tool in real model debugging tasks. We have installed the tool on ten modelers' workstations, and are collecting topology, navigation, and cue data on users' modeling sessions. These data will allow us to validate the patch, navigation, and abstract syntax results above, as well as to collect detailed "in the wild" information foraging data on a domain markedly different from the Java navigation data in Objectives 5a and 6a.

3. Publications

We are proud to have produced 12 publications under this grant, which have already been cited 87 times.

2009

1. [Lawrance et al. 2009] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, and Kyle Rector, 2009. How people debug, revisited: An information foraging theory perspective, IBM Technical Report RC24783.
2. [Lawrance 2009] Joseph Lawrance, 2009. Information Foraging in Debugging. Ph.D. Dissertation, Oregon State University.

2010

3. [Lawrance et al. 2010a] Joseph Lawrance, Margaret Burnett, Rachel Bellamy, Christopher Bogart, and Calvin Swart, 2010. Reactive information foraging for evolving goals, *ACM Conference on Human Factors in Computing Systems*, April 2010, 25-34.

4. [Burnett 2010] Margaret Burnett, The Future of software engineering: Enhancing human expertise in tackling software quality (position paper), *ACM FSE/SDP Workshop on the Future of Software Engineering Research*, 75-76.
5. [Lawrance et al. 2010b] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott Fleming. How programmers debug, revisited: An information foraging theory perspective, *IEEE Transactions on Software Engineering*, (accepted 2010, available in IEEE Digital Library, to appear in print).
6. [Bogart et al. 2010] Christopher Bogart, Margaret Burnett, Scott Douglass, David Piorkowski, Amber Shinsel, 2010. Does my model work? Evaluation abstractions of cognitive modelers, *IEEE Symposium on Visual Languages and Human-Centric Computing*, 49-58.

2011

7. [Piorkowski et al. 2011] David Piorkowski, Scott D. Fleming, Christopher Scaffidi, Liza John, Christopher Bogart, Bonnie E. John, Margaret Burnett, and Rachel Bellamy, 2011. Modeling programmer navigation: A head-to-head empirical evaluation of predictive models, *IEEE Symposium on Visual Languages and Human-Centric Computing,* 109-116.
8. [Ko et al. 2011] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Joseph Lawrance, Chris Scaffidi, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck, 2011. The state of the art in end-user software engineering, *ACM Computing Surveys* 43(3), Article 21.

2012

9. [Piorkowski et al. 2012a] David Piorkowski, Scott D. Fleming, Christopher Scaffidi, Christopher Bogart, Margaret M. Burnett, Bonnie E. John, Rachel K. E. Bellamy, Calvin Swart, 2012. Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers. *ACM Conference on Human Factors in Computing Systems*, 1471-1480.
10. [Bogart et al. 2012] Christopher Bogart, Margaret Burnett, Scott Douglass, Hannah Adams, Rachel White, 2012. Designing a debugging interaction language for cognitive modelers: An initial case study in Natural Programming Plus. *ACM Conference on Human Factors in Computing Systems*, 2469-2478.
11. [Fleming et al. 2012] Scott Fleming, Christopher Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, Irwin Kwan, 2012. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Transactions on Software Engineering Methodology*, Volume 22, Issue 2 (to appear).
12. [Piorkowski et al. 2012b] David Piorkowski, Scott Fleming, Irwin Kwan, Margaret Burnett, Christopher Scaffidi, Rachel Bellamy, Joshua Jordahl. The whats and hows of programmers' foraging diets (under review).

References (outside our own publications)

[Baeza-Yates et al. 1999] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison Wesley Longman, 1999.

[Chi et al. 2001] E. Chi, P. Pirolli, K. Chen, and J. Pitkow, 2001. Using information scent to model user information needs and actions and the Web. In *ACM Conference on Human Factors in Computing Systems*, 490-497.

[Fu and Pirolli 2007] W.-T. Fu, P. Pirolli. 2007. SNIF-ACT: a cognitive model of user navigation on the world wide web. *Human-Computer Interaction*, *22*, 4, 355–412.

[Grigoreanu et al. 2010] Grigoreanu, V., Burnett, M. and Robertson, G. 2010. A strategy-centric approach to the design of end-user debugging tools. *ACM Conference on Human Factors in Computing Systems*, 713-722.

[Pirolli 1997] P. Pirolli, 1997. Computational models of information scent-following in a very large browsable text collection. *ACM Conference on Human Factors in Computing Systems*, 3-10.

[Pirolli and Card 1998] P. Pirolli, S. Card. 1998. Information foraging models of browsers for very large document spaces. *Proceedings of the Working Conference on Advanced Visual Interfaces*. 83–93.

[Pirolli and Card 1999] P. Pirolli, S. Card, 1999. Information foraging. *Psychological Review*, *106*, 4, 643-675.

[Sillito et al. 2006] J. Sillito, G. Murphy, K. De Volder, 2006. Questions programmers ask during software evolution tasks. *ACM Foundations of Software Engineering*, 23–34.

Appendix A: Personnel Supported By Grant

The following individuals were funded using resources from this grant.

- Hannah Adams (undergraduate research assistant)
- Forrest Bice (high school intern/undergraduate research assistant)
- Chris Bogart (Ph.D. student)
- Margaret Burnett (Professor, PI)
- Scott D. Fleming (postdoctoral scholar)
- Irwin Kwan (postdoctoral scholar)
- Joseph Lawrance (Ph.D. student)
- Chris Scaffidi (Assistant Professor)
- David Piorkowski (Masters student)
- Kyle Rector (undergraduate research assistant)
- Rachel White (high school intern/undergraduate research assistant)

Appendix B: Summary of Lawrance Thesis

Title: Information Foraging in Debugging

An Abstract for the Dissertation of Joseph Lawrance for the degree of Doctor of Philosophy in Computer Science, Oregon State University.

Programmers spend a substantial fraction of their debugging time by navigating through source code, yet little is known about how programmers navigate. With the continuing growth in size and complexity of software, this fraction of time is likely to increase, which presents challenges to those seeking both to understand and address the needs of programmers during debugging.

Therefore, we investigated the applicability of a theory from another domain, namely information foraging theory, to the problem of programmers' navigation during software maintenance. The goal was to determine the theory's ability to provide a foundational understanding that could inform future tool builders aiming to support programmer navigation.

To perform this investigation, we first defined constructs and propositions for a new variant of information foraging theory for software maintenance. We then operationalized the constructs in different ways and built three executable models to allow for empirical investigation. We developed a simple information-scent-only model of navigation, a more advanced model of programmer navigation, named Programmer Flow by Information Scent (PFIS), which accounts for the topological structure of source code, and PFIS2, a refinement of PFIS that maintains an up-to-date model of source code on the fly and models information scent even in the absence of explicit information about stated goals.

Complete thesis is available at:
http://ir.library.oregonstate.edu/xmlui/handle/1957/11994